计算机组成

Cache

高小鹏

北京航空航天大学计算机学院

存储层次的动机

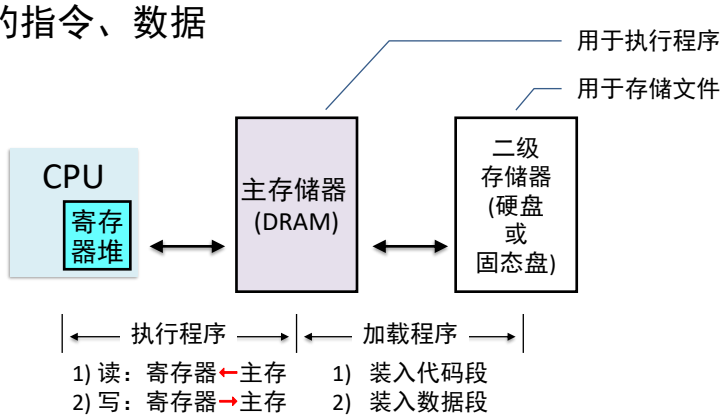
提纲

- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

3

程序员眼中的存储体系1/2

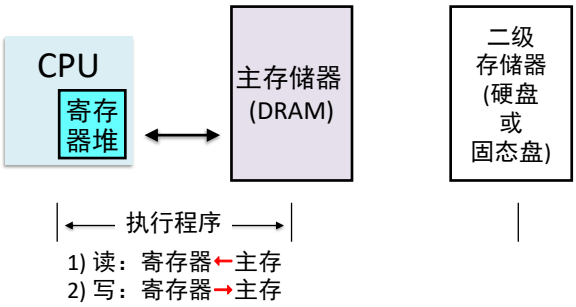
- 二级存储器
  - 可执行程序（word.exe）、数据文件（电影）
- 主存储器
  - 程序的指令、数据



4

程序员眼中的存储体系2/2

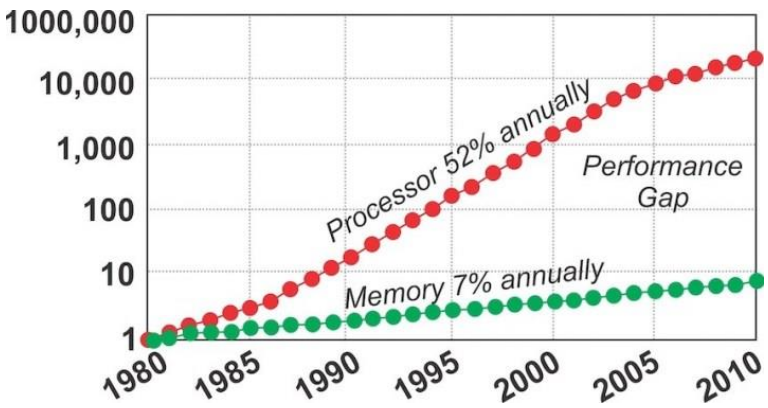
- 问题：这个结构执行程序的性能如何？



5

存储墙

- 从1980-2010，处理器性能增长速率远超主存储器
- 主存储器的性能已经成为计算机系统的性能瓶颈了



图片资料来源：https://www.rankred.com/worlds-fastest-optical-ram/

※存储墙(Memory Wall)：意指处理器与主存储墙之间的巨大性能差距　计算机组成与实现

C代码的功能？ 1/2

```
char *p ;
int  length = 0;

   ⋮

1 while (*p++)
2     length++ ;
```


C代码的功能？ 2/2

- 答案：计算字符串的长度

```
char *p ;
int  length = 0;

   ⋮

1 while (*p++)
2     length++ ;
```

1）从p处读取字符
2）p后移
3）若字符不为0，则继续循环，否则退出

length加1

汇编代码1/3

- 处理1个字符：5条指令
  - ①读字符；②比较；③移动指针；④长度加1；⑤重复

假设：$s0←→p，$s1←→length；$t0←→读入的字符

		Loop:	
while (*p++)	1	lb $t0, 0($s0)	读入字符
length++;	2	beq $t0, $zero, End	若为0则退出
	3	addi $s0, $s0, 1	指针后移
	4	addi $s1, $s1, 1	长度加1
	5	j Loop	重复执行
		End:	

9

汇编代码2/3

- 处理1个字符：5条指令
  - ①读字符；②比较；③移动指针；④长度加1；⑤重复

假设：$s0←→p，$s1←→length；$t0←→读入的字符

		Loop:	
while (*p++)	1	lb $t0, 0($s0)	读入字符
length++;	2	beq $t0, $zero, End	若为0则退出
	3	addi $s0, $s0, 1	指针后移
	4	addi $s1, $s1, 1	长度加1
	5	j Loop	重复执行
		End:	

基本结论：处理*N*个字符约需要*5N*指令

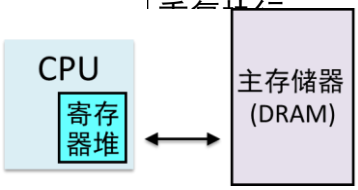
10

汇编代码3/3

- 假设：CPU能够1个时钟周期执行1条指令
- 问题：5N条指令真的只需要5N个时钟周期？

假设：$s0←→p，$s1←→length；$t0←→读入的字符

```
while (*p++)          Loop:
  length++;      1      lb   $t0, 0($s0)      读入字符
                 2      beq  $t0, $zero, End  若为0则退出
                 3      addi $s0, $s0, 1      指针后移
                 4      addi $s1, $s1, 1      长度加1
                 5      j    Loop
                 End:
```



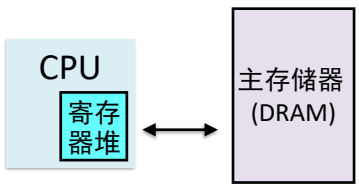
11

存储器的典型性能参数

- 延迟：读/写1个字节需要的时间
  - CPU寄存器：0.5个时钟周期
  - 主存储器：100个时钟周期

```
   Loop:
1    lb   $t0, 0($s0)
2    beq  $t0, $zero, End
3    addi $s0, $s0, 1
4    addi $s1, $s1, 1
5    j    Loop
   End:
```

DRAM特点
读取连续数据时，第1个数据等待时间很长，后续极快



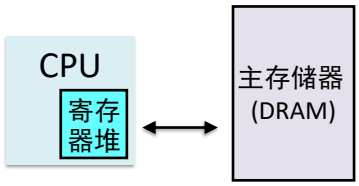
	CPU寄存器	主存储器
性能(周期)	0.5	100
容量(字节)	100	M

12

更接近真实的性能估计1/2

- 处理1个字符：104个时钟周期！
  - 指令1：涉及访问主存，需要100个时钟周期
  - 指令2~5：仅与寄存器相关，共需要4个时钟周期

```
    Loop:
1     lb    $t0, 0($s0)        100
2     beq   $t0, $zero, End    1
3     addi  $s0, $s0, 1        1
4     addi  $s1, $s1, 1        1
5     j     Loop               1
    End:
```



	性能(周期)	0.5	100
	容量(字节)	100	M

13

更接近真实的性能估计2/2

- 处理1个字符：104个时钟周期！
  - 指令1：涉及访问主存，需要100个时钟周期
  - 指令2~5：仅与寄存器相关，共需要4个时钟周期

```
    Loop:
1     lb    $t0, 0($s0)        100
2     b
3     a
4     a
5     j
    End
```

基本结论1：处理*N*个字符约需要*5N*指令

CPU理想执行时间：*5N*个时钟周期

基本结论2：CPU约执行*104N*个时钟周期

	性能(周期)	0.5	100
	容量(字节)	100	M

14

局部性原理

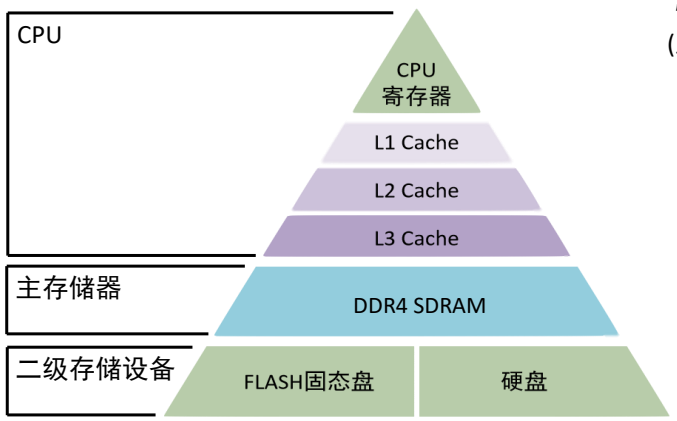
- 时间局部性
  - 如果某个存储单元被引用，那么不久它可能再次被引用
- 空间局部性
  - 如果某个存储单元被引用，那么它相临的存储单元可能很快会被引用
- 局部性的成因
  - 顺序执行：绝大多数指令是顺序执行的
  - 循环结构：指令会被重复访问
  - 数据结构：数组、结构等

```
while (*p++)
  length++ ;
```

15

计算机的存储层次1/2

- 目标：整体性能接近寄存器，成本接近硬盘
- 特点：性能、容量、成本具有数量级差距

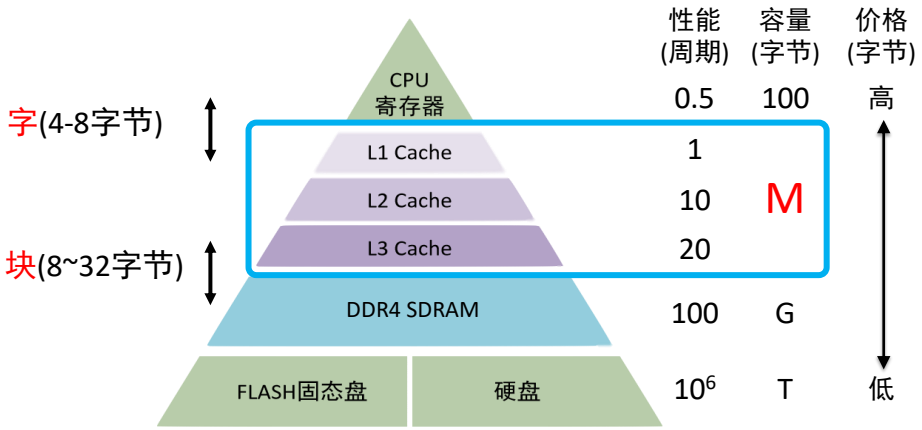


	性能 (周期)	容量 (字节)	价格 (字节)
	0.5	100	高
	1		
	10	M	
	20		
	100	G	
	10^6	T	低

SRAM~Static Random Access Memory，静态存储器；SDRAM~Synchronous Dynamic RAM，动态存储器

计算机的存储层次2/2

- Cache是解决CPU-主存性能匹配的关键
  - 小容量、高性能
  - 充分利用了局部性原理



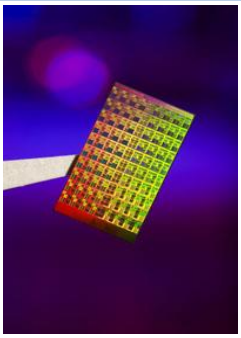
17

多核的出现

- 多核是无奈的选择
- 单核性能提升方法1：增加主频
  - 处理器性能 = 主频 × IPC
  - 处理器功耗正比于主频的三次方
- 单核性能提升方法2：增加晶体管
  - 晶体管×2，性能×$\sqrt{2}$

百核/千核：未来已经可见！

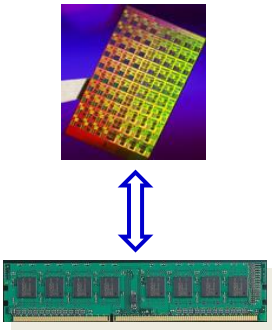
- Intel 80核
  - 80个简单核
  - 2个浮点部件/core
  - 片内2维互联网络
  - 1亿晶体管



频率	电压	功耗	带宽	性能
3.16 GHz	0.95 V	62W	1.62 Terabits/s	1.01 Teraflops
5.1 GHz	1.2 V	175W	2.61 Terabits/s	1.63 Teraflops
5.7 GHz	1.35 V	265W	2.92 Terabits/s	1.81 Teraflops

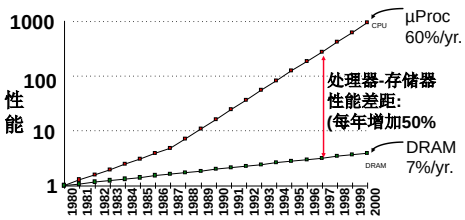
Intel 80核 vs. DDR3 800

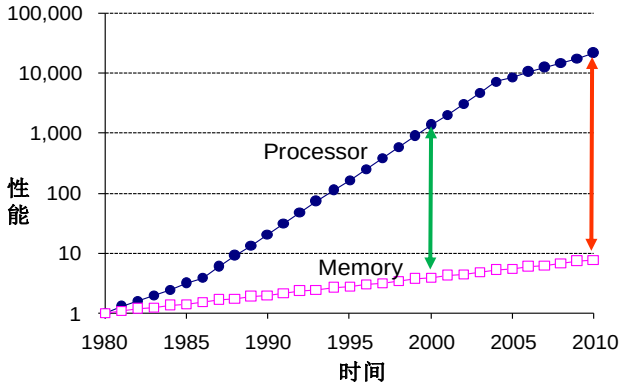
- 1次浮点运算需2个浮点数
  - 假设1：2个源操作数都来自DRAM
  - 假设2：不回写计算结果
- 单精度浮点比例：640
  - 1T×2 / (12.8G/4) = 640
- 双精度浮点比例：1280
  - 1T ×2/ (12.8G/8) = 1280
- 意味着：如果要用一条DDR3让80核吃饱，则数据进入后必须运行1280次才能出CPU！



多核导致性能差距进一步恶化

- 2000年：300～400倍
- 2010年：1000倍+





我眼中的Cache

- 构思：发掘局部性
  - 时间局部性、空间局部性
  - 2-8原则
- 功效：四两拨千斤
  - 结构相对简单、性能改善巨大
- 影响：致深致远
  - CPU：L1、L2、L3
  - OS：TLB、磁盘缓存
  - 网络：映射表
  - 应用：。。。

提纲

- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

23

主存

- 主存抽象为二维数组
  - 主存地址：数组下标
- 主存地址：6位
  - 2^6=64
- 主存块数：16块
  - 64/4=16

主存容量	64B
Cache容量	16B
块大小	4B

11	10	01	00	
3	2	1	0	0000
7	6	5	4	0001
11	10	9	8	0010
15	14	13	12	0011
19	18	17	16	0100
23	22	21	20	0101
27	26	25	24	0110
31	30	29	28	0111
35	34	33	32	1000
39	38	37	36	1001
43	42	41	40	1010
47	46	45	44	1011
51	50	49	48	1100
55	54	53	52	1101
59	58	57	56	1110
63	62	61	60	1111

Offset：偏移

10进制	2进制
39	100111
38	100110
37	100101
36	100100

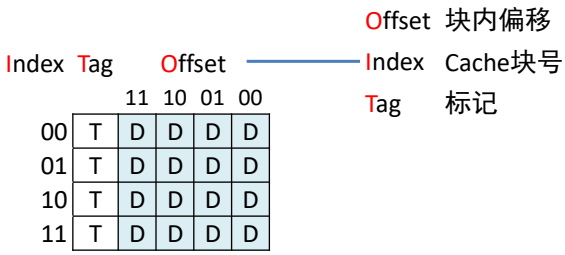
主存块号

24

Cache

- Cache地址：4位
  - 2^4=16
- Cache块数：4块
  - 16/4=4

主存16块，Cache4块
必须建立16块到4块的对应关系

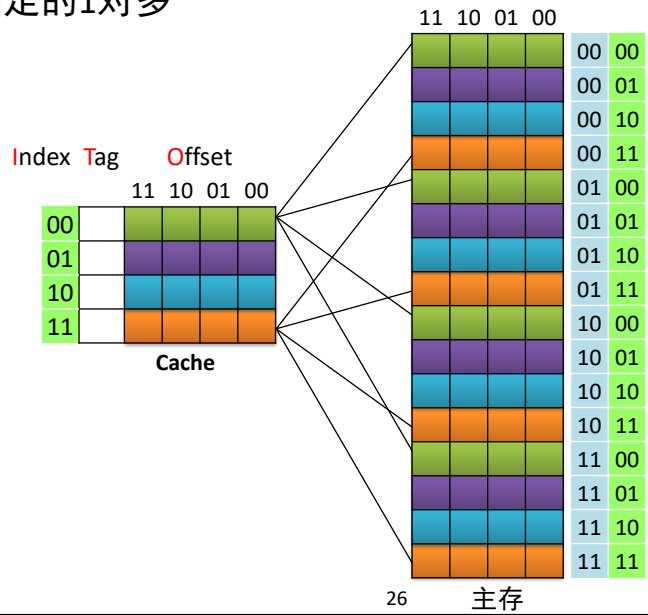


主存容量	64B
Cache容量	16B
块大小	4B

25

主存块与Cache块的映射关系

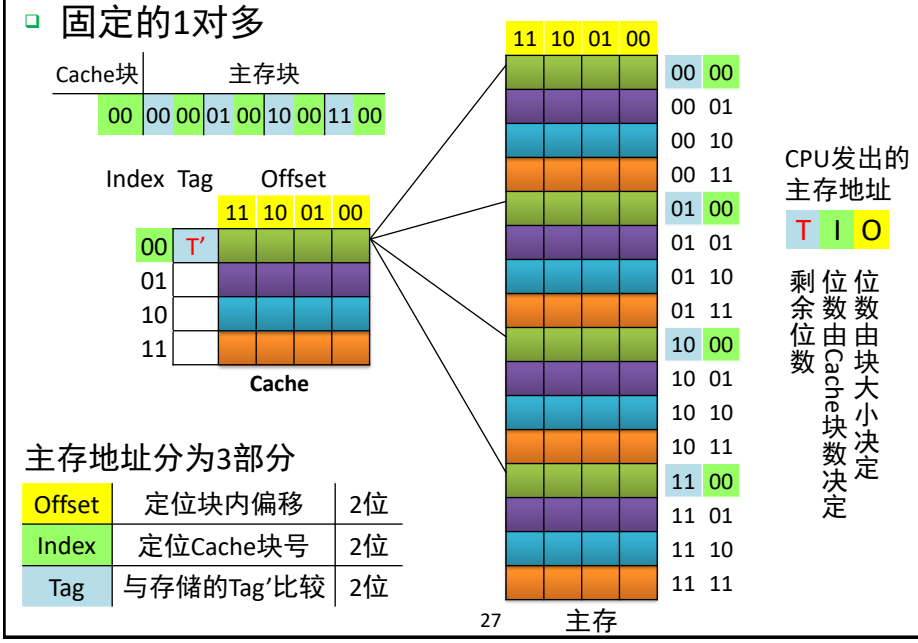
- 固定的1对多



26 主存

主存地址结构：TIO

- 固定的1对多

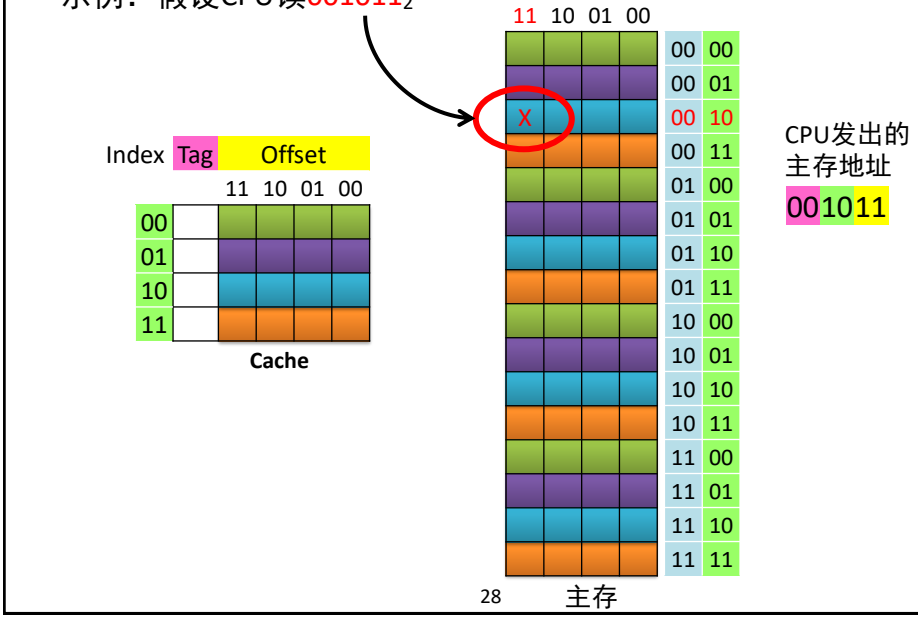


主存地址分为3部分

Offset	定位块内偏移	2位
Index	定位Cache块号	2位
Tag	与存储的Tag'比较	2位

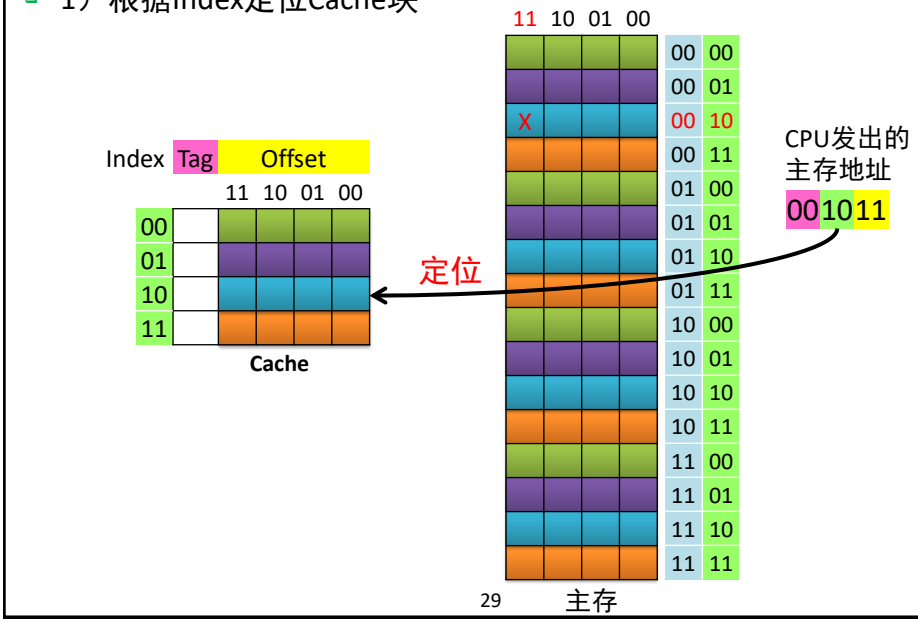
从主存单元到Cache单元1/4

- 示例：假设CPU读001011_2



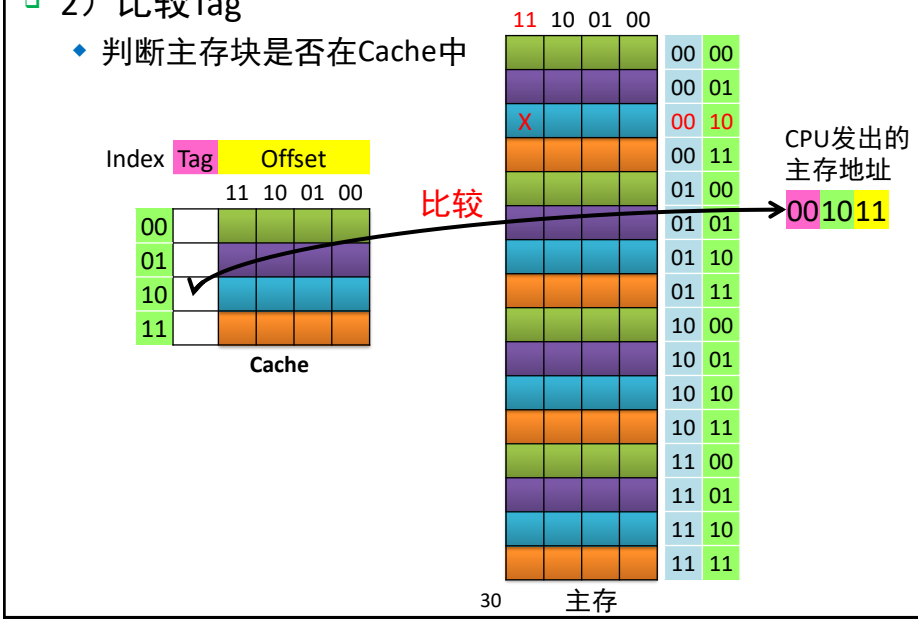
从主存单元到Cache单元2/4

- 1）根据Index定位Cache块



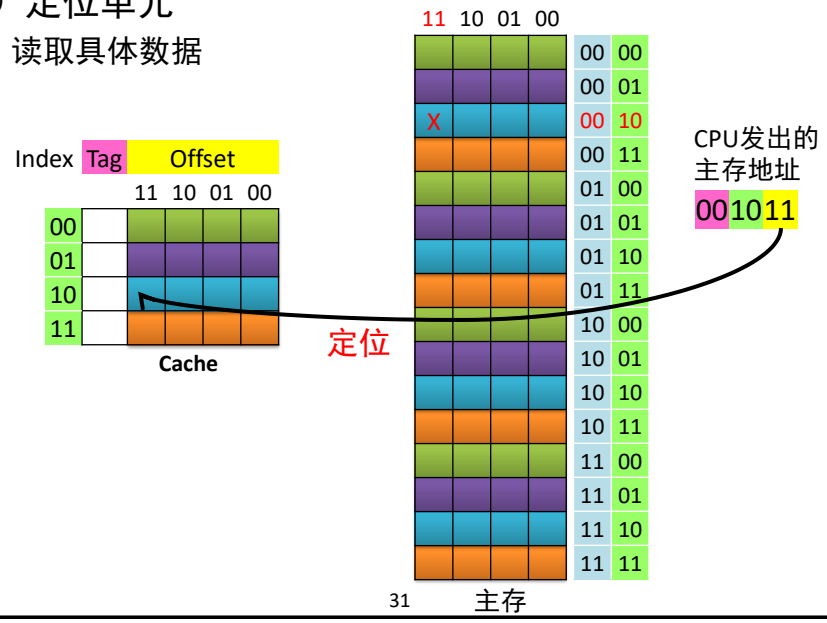
从主存单元到Cache单元3/4

- 2）比较Tag
  - 判断主存块是否在Cache中



从主存单元到Cache单元4/4

- 3）定位单元
  - 读取具体数据



TIO的一般性计算方法

- 假设：主存地址位数为A

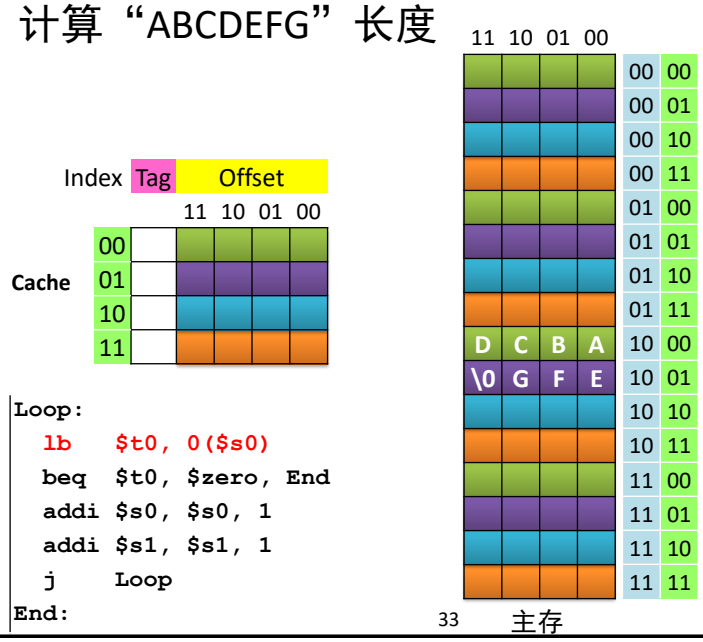
Offset	$\log_2$ 块大小
Index	$\log_2$ $Cache$块数
Tag	A-I-O

- 示例：32位主存地址，Cache 256KB，每块16B
  - Offset：$\log_2 16 = 4$位
  - Index：256KB/16B=16K块，$\log_2 16K$ =14位
  - Tag：32-14-4=14位

32

示例：Cache对性能的影响

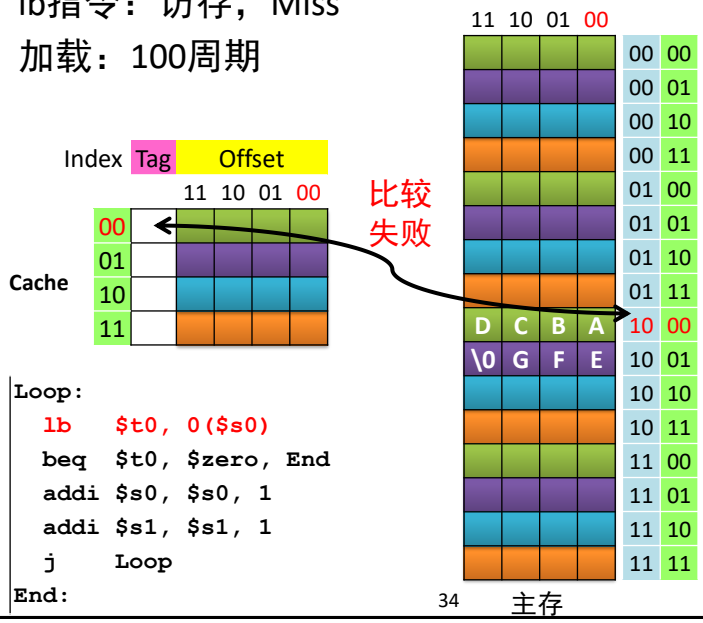
- 计算“ABCDEFG”长度




```
Loop:
1   lb   $t0, 0($s0)
2   beq  $t0, $zero, End
3   addi $s0, $s0, 1
4   addi $s1, $s1, 1
5   j    Loop
End:
```

字符A 1/2

- lb指令：访存，Miss
- 加载：100周期

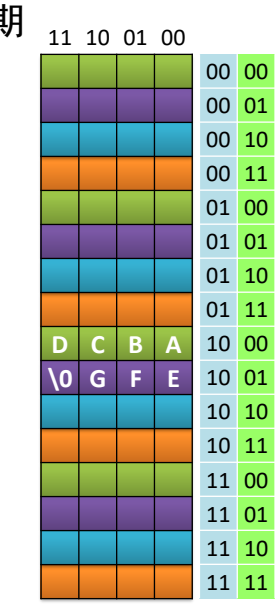



```
Loop:
1   lb   $t0, 0($s0)
2   beq  $t0, $zero, End
3   addi $s0, $s0, 1
4   addi $s1, $s1, 1
5   j    Loop
End:
```

字符A2/2

- 其余指令：无访存，4周期
- 累计：104周期


```
   Loop:
1    lb   $t0, 0($s0)
2    beq  $t0, $zero, End
3    addi $s0, $s0, 1
4    addi $s1, $s1, 1
5    j    Loop
   End:
```



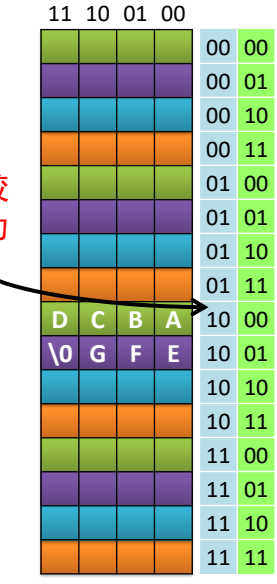
35 主存

字符B、C、D

- lb指令：均Hit
- 共计：5x3=15周期

比较成功

```
   Loop:
1    lb   $t0, 0($s0)
2    beq  $t0, $zero, End
3    addi $s0, $s0, 1
4    addi $s1, $s1, 1
5    j    Loop
   End:
```

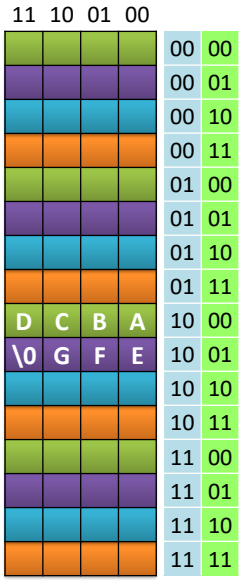


36 主存

字符E、F、G

- E：Miss，104周期
- F、G：Hit，10周期


```
Loop:
1   lb   $t0, 0($s0)
2   beq  $t0, $zero, End
3   addi $s0, $s0, 1
4   addi $s1, $s1, 1
5   j    Loop
End:
```



37 主存

性能统计

循环	字符	命中/缺失	时间(周期)
1	A	缺失	104
2、3、4	B、C、D	命中	5x3=15
5	E	缺失	104
6、7	F、G	命中	5x2=10
			233

无cache：104x7=**728**时钟周期（*104N*）

有cache：**233**时钟周期

38

直接映射cache的容量

- 一个cache块不仅包含数据，还包含tag、valid等
- 数据位：8×2^{O}位
- Tag位：主存地址位数–Index位数–Offset位数
- Valid位：1位，指示cache块是否包含有效数据
- Cache总容量 = cache块数×(数据位 + Tag位数 + Valid位数)
  = $2^{Index}\times(8\times2^{Offset}+Tag+1)$位

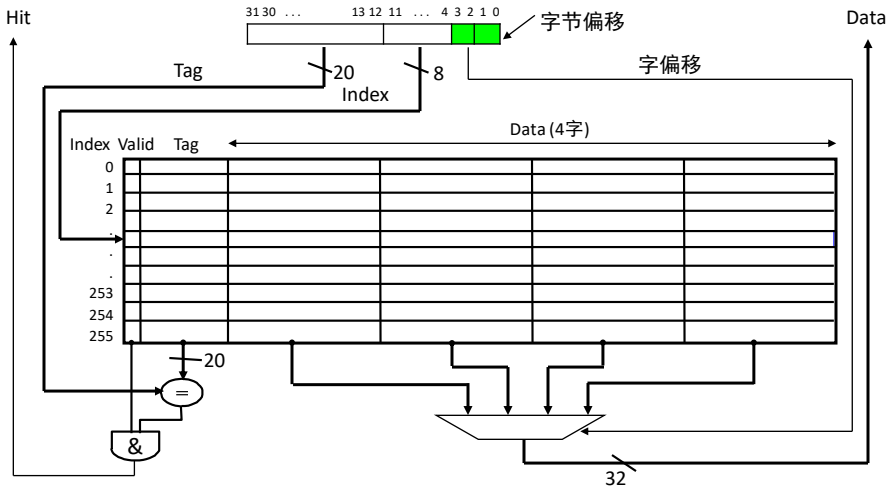
Cache示例

- 基本参数
  - 主存地址空间16MB，cache容量为16KB，cache块为4W
- 主存地址TIO结构
  - 主存地址位数：$\log_2(16M)=24$位
  - Offset：4W即16B，offset位数为$\log_2(16)=4$位
  - Index：cache容量/块大小=16KB/16B=1K，index位数=$\log_2(1K)=10$位
  - Tag：主存地址位数 – index – offset = 24 – 10 – 4= 10位

- Cache容量= $2^{Index}\times(8\times2^{Offset}+Tag+1)$
  = $2^{10}\times(8\times2^{4}+10+1)$ 位

直接映射cache内部结构

- 参数：32位主存地址，cache容量为4KB，块大小为16B



Cache术语

- 命中（Hit）
  - Tag比较成功
  - 主存块在Cache中，可快速读取
- 缺失 （Miss）
  - Tag比较失败
  - 主存块不在Cache中，必须从主存中读取该块
- Cache块替换
  - 当前Cache块存储的不是CPU请求的主存块
  - 必须从主存中读取该主存块，并将当前Cache块替换

Cache术语

- Cache效率参数：最大化命中，最小化缺失
  - 命中率（Hit Rate，HR）
    - 对于一段程序，命中次数占总存储访问次数的比例
  - 缺失率（Miss Rate，MR）
    - 对于一段程序，缺失次数占总存储访问次数的比例
- HR与MR恒满足

$$HR + MR = 1$$

Cache术语

- Cache延迟参数
  - 命中时间（Hit time，HT）
    - 访问cache的时间（包含了tag比较时间）
  - 缺失代价（Miss penalty，MP）
    - 从下一层存储器读入一个cache块的时间

Cache失效的原因（3C）

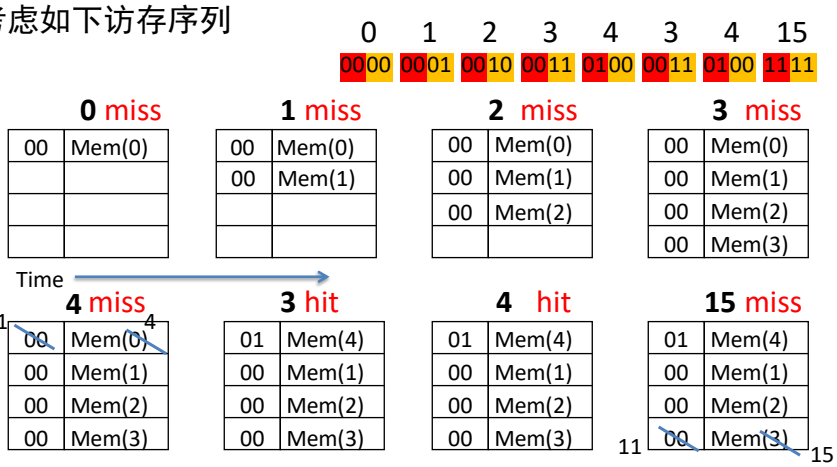
- 强制：Compulsory (冷启动或者进程切换导致的首次访问)
  - 无法避免首次访问cache块时的缺失
  - 如果程序运行时间很长，则可以忽略
- 容量：Capacity
  - Cache不可能包含程序要访问的所有主存块
- 冲突：Conflict（collision）
  - 多个主存块映射到同一个cache块

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

Cache结构与性能

- 存储参数：主存16B，cache容量4B，cache块为1B
  - TIO：2-2-0
- 考虑如下访存序列



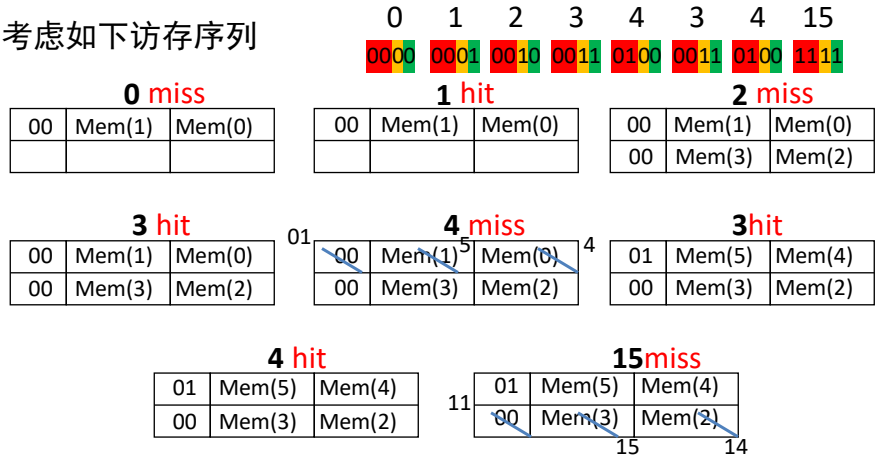
- 8个请求，6次缺失（HR = 0.25， MR = 0.75）

Cache结构与性能

- 存储参数：主存16B，cache容量4B，cache块为2B

利用局部性
增加块尺寸

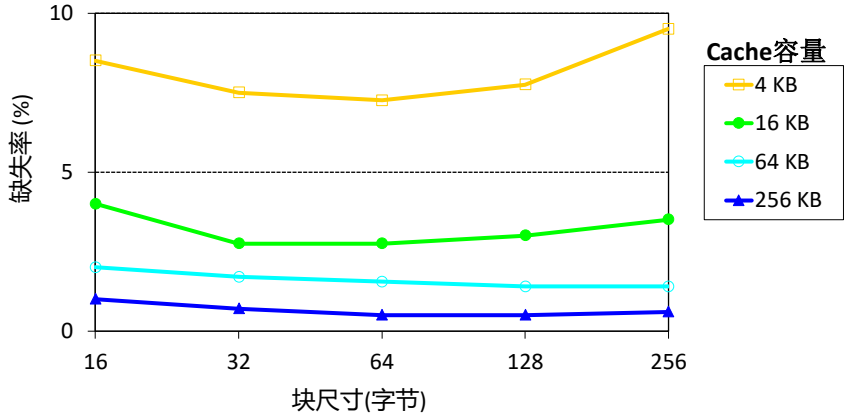
  - TIO：2-1-1
- 考虑如下访存序列



- 8个请求，4次缺失（HR = 0.5，MR = 0.5）

Cache结构与性能

- Cache性能的浴盆曲线
  - 随着块尺寸的增加，缺失率下降：块尺寸增加，块上的命中得到提升
  - 若块尺寸不断增长，缺失率上升：冲突概率增加，导致频繁替换



计算机组成与实现

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

Cache一致性问题

- 由于主存与cache都可能存储着数据，因此就产生了一致性问题
- 假设CPU连续2次访问主存X单元

时间	事件	Cache内容	位置X的主存内存
0			1
1	CPU读X	1	1
2	CPU将0写入X	0	1

Cache的读和写

- Cache对于命中和缺失的处理策略是独立的
- 假设使用分离的指令cache(I$)和数据cache(D$)
  - 读：I$与D$都有读
    - I$：取指令
    - D$：读数据
  - 写：只对D$有效

Cache命中1/2

- 读命中(I$与D$)
  - 性能最高；这是最理想的cache访问模式
- 写命中(D$)
  - 1）写通(Write-Through)：总是同时写cache与主存
    - cache与主存总是一致的
    - 慢！写的性能就是主存的性能
    - 改善措施：可以与cache平行部署一个Write Buffer

Cache命中2/2

- 读命中(I$与D$)
  - 性能最高；这是最理想的cache访问模式
- 写命中(D$)
  - 2）写回(Write-Back)：只写cache，当cache块需要被替换时再更新主存
    - 这意味着cache与主存之间是可以不一致的
    - 命中在同一cache块的多个写被收集起来了，提高了写的效率
    - Dirty位：每个cache块需要增加一个dirty为，用于记录是否被写入过，即在替换时需要写回到主存

处理Cache缺失1/2

- 缺失代价：随着block变大而变大
- 读缺失（I$和D$）
  - ①流水线暂停；②从主存读入block；③写入cache；④发送word至流水线；⑤流水线恢复执行
- 写缺失（D$）
  - 写分配（write allocate）： ①流水线暂停；②从主存读入block；③写入cache；④执行写命中
    - 无论写通还是写回，写分配均有效
    - 可以确保在写缺失后，cache有最新数据

处理Cache缺失2/2

- 缺失代价：随着block变大而变大
- 读缺失（I$和D$）
  - ①流水线暂停
  - ②从主存读入block
  - ③写入cache
  - ④发送word至流水线
  - ⑤流水线恢复执行
- 写缺失（D$）
  - 非写分配（No-write allocate）：不写cache而直接写主存
    - cache不包含最新数据
    - 但主存包含最新数据

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

Cache的AMAT

- 影响cache性能的两大主要因素
  - 缺失率(MR)与缺失代价(MP)
- 平均存储器访问时间(Average Memory Access Time，AMAT)：综合考虑命中和缺失后的存储器访问的平均延迟

平均存储器访问时间 = 命中时间 + 缺失率×缺失代价

(缩写：AMAT = HT + MR × MP)

Cache的AMAT

- Cpu参数：200ps/clock，MP为50时钟周期，MR为2%，HT为1个时钟周期

  AMAT = HT + MR × MP = 1 + 2% × 50 = 2时钟周期 = 400ps

- 下面哪个方案改进效果更显著？
  - 190ps/clock　　380 ps
  - MP为40时钟周期　　360 ps
  - MR为1.5%　　350 ps

Cache容量增加，性能一定提升吗？

- 当增加cache容量后，cache的AMAT会受哪些参数影响？
  - HR会↑
  - HT会↑：cache越小，HT越快
  - 注意：有时对于大容量cache来说，HT增加会抵消掉HR带来的收益。因此大容量cache不总是比小容量cache性能更好

Cache对CPI的影响

- CPU性能

$$CPU\text{时间} = \text{指令数} \times \text{CPI} \times \text{时钟周期时间}$$

- 考虑存储器访问延迟的CPI

$$CPI_{\text{stall}} = CPI_{\text{base}} + \text{平均存储延迟周期}$$

- 存储延迟周期

$$\text{存储延迟周期} = \frac{\text{访存次数}}{\text{指令}} \times MR \times MP$$

CPI示例

- CPU性能参数：
  - CPI_{base}为1
  - MP代价为100周期（对于I$与D$相同）
  - load/store指令频度为36%
  - I$缺失率为2%，D$缺失率为4%
- 关键点：平均每条指令访问I$与D$各多少次？

CPI示例

- CPU性能参数：
  - CPI_{base}为1
  - MP代价为100周期（对于I$与D$相同）
  - load/store指令频度为36%
  - I$缺失率为2%，D$缺失率为4%
- 指令访存次数：1次指令存储，36%次数据存储
- 访存延迟周期：指令访存延迟+数据访存延迟

$$\text{访存延迟周期} = 100\% \times 2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$$

$$CPI_{stall} = 1 + 3.44 = 4.44$$

如果D$缺失率减少1%，效果如何？

3C与cache设计

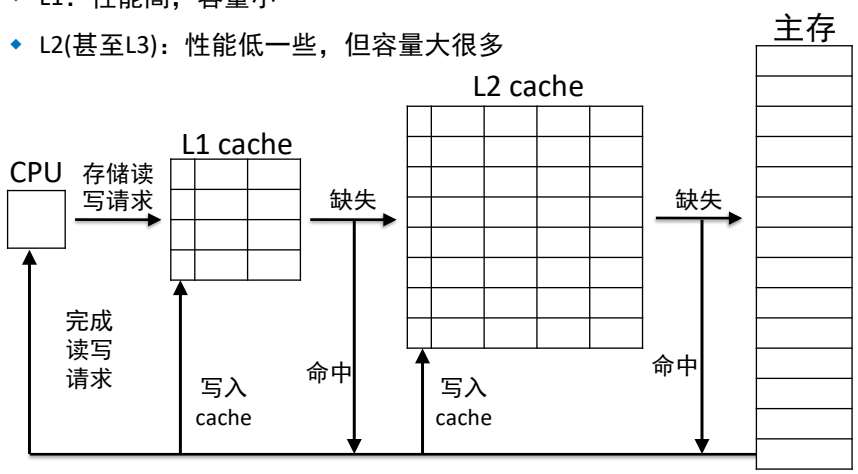
- Compulsory
  - 增加块容量
    - 副作用：导致MP↑；块越大，MR也有可能会增加
- Capacity
  - 增加cache容量
    - 副作用：可能导致HT↑
- Conflict
  - 增加cache容量
  - 增加相联度
    - 副作用：可能导致HT↑

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

多级cache架构

- 矛盾：存储器越快，单位成本越高
- 方案：cache也采用多层架构
  - L1：性能高，容量小
  - L2(甚至L3)：性能低一些，但容量大很多



多级cache的AMAT

- AMAT = L1 HT + L1 MR × L1 MP
  - 现在L1 MP依赖于L2的性能
- L1 MP = L2 HT + L2 MR × L2 MP
  - 如果有更多层次，则继续迭代

$$MP_i = HT_{i+1} + MR_{i+1} \times MP_{i+1}$$

  - 最后一层就是主存的访问时间
- 对于2级cache

$$AMAT = L1\ HT + L1\ MR \times (L2\ HT + L2\ MR \times L2\ MP)$$

多级cache的AMAT示例

- Cpu参数：L1 HT为1个周期，L1 MR为2%，L2 HT为5个周期，L2 MR为5%，主存访问时间为100个周期
  - 假设L1是一体的而非分离的
- 没有L2$：

$$AMAT_1 = 1 + 0.02 \times 100 = 3$$

- 有L2$:

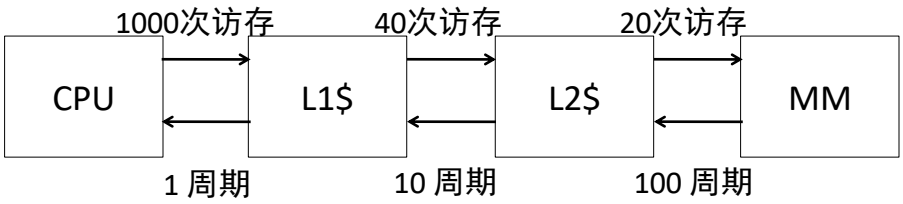
$$AMAT_2 = 1 + 0.02 \times (5 + 0.05 \times 100) = 1.2$$

局部缺失率 vs. 全局缺失率

- 局部缺失率（Local miss rate）：对于某层cache的访问而产生的缺失比例
  - 例如：L2$局部缺失率 = L2$缺失次数/L1$缺失次数
- 全局缺失率（Global miss rate）：对于所有各级cache的访问而产生的缺失比例
  - 这事实上是整个存储层次的总体性能参数
- 对于N级cache来说，全局缺失率有两种计算方法：
  - 方法1：$\text{全局缺失率} = \prod L_i^{\text{局部缺失率}}$
  - 方法2：全局缺失率 = L_n缺失次数/L1访问次数
- 根据定义可知，全局MR ≤ 任意的局部MR

2级cache的局部缺失率与全局缺失率

- 对于1000次访存
  - L1$产生40次缺失。L1$的局部缺失率？ 0.04
  - L2$产生20次缺失。L2$的局部缺失率？ 0.5
  - 全局缺失率？ 0.02



更全面的性能计算公式

- 对于2级cache，我们知道：

$$MR_{global} = L1\ MR \times L2\ MR$$

- AMAT：

$$AMAT = L1\ HT + L1\ MR \times (L2\ HT + L2\ MR \times L2\ MP)$$

$$= L1\ HT + L1\ MR \times L2\ HT + MR_{global} \times L2\ MP$$

- CPI：

$$CPI_{\text{stall}} = CPI_{base} + \frac{\text{访存次数}}{\text{指令}} \times L1MR \times (L1\ MP + L2\ MR \times L2\ MP)$$

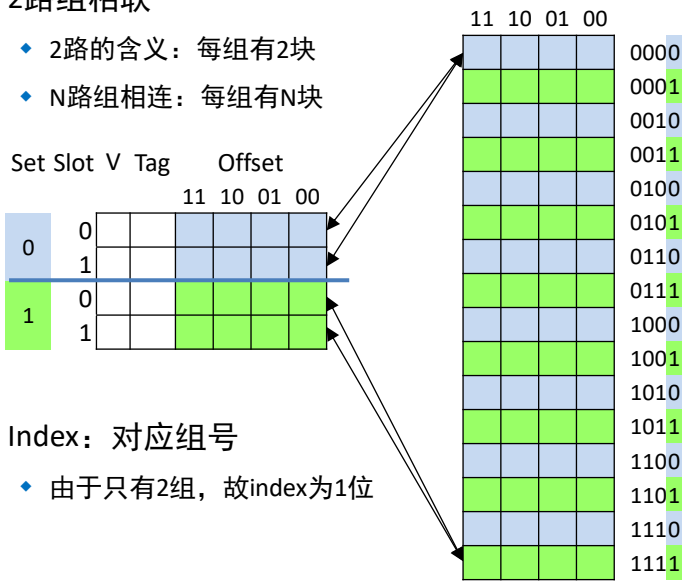
$$CPI_{\text{stall}} = CPI_{base} + \frac{\text{访存次数}}{\text{指令}} \times (L1MR \times L1\ MP + MR_{Global} \times L2\ MP)$$

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

组相联cache示例

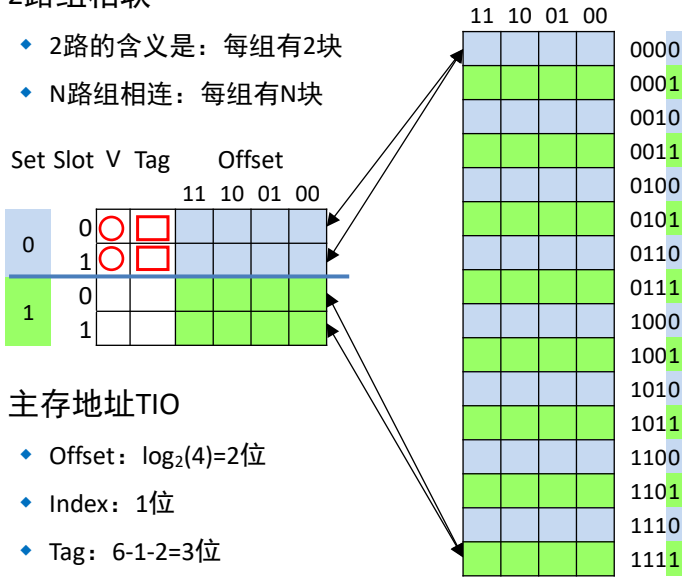
- 2路组相联
  - 2路的含义：每组有2块
  - N路组相连：每组有N块



- Index：对应组号
  - 由于只有2组，故index为1位

组相联cache示例

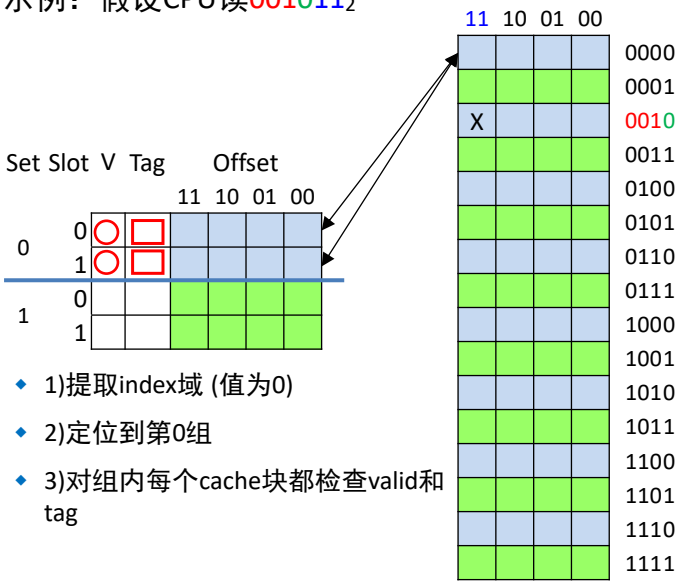
- 2路组相联
  - 2路的含义是：每组有2块
  - N路组相连：每组有N块



- 主存地址TIO
  - Offset：$\log_2(4)$=2位
  - Index：1位
  - Tag：6-1-2=3位

组相联cache访问机制

- 示例：假设CPU读001011_2



- 1)提取index域 (值为0)
- 2)定位到第0组
- 3)对组内每个cache块都检查valid和tag

组相联cache示例

- 假设：主存地址位数为A，分解为TIO结构

- Index计算方式发生变化
- Tag和offset的计算方式不变

Offset $\log_2$ 块大小

Index $\log_2$ $Cache$组数

Tag A-I-O

- 示例：32位主存地址；Cache为256KB，每块16B，4路组相联
  - Offset：$\log_2 16 = 4$位
  - Index：256KB/16B=16K块，16K块/4路=4K组，$\log_2 4K$ =12位
  - Tag：32-12-4=16位

Question: What is the TIO breakdown for the following cache?

- 32-bit address space
- 32 KiB 4-way set associative cache
- 8 words/block

	T	I	O
☐	21	8	3
☐	19	8	5
☐	19	10	3
☐	17	10	5

77

组相联读写示例

- 假设cache只有4B，每块为1B
  - 各cache块初始均为空
- 假设cpu访存序列：0，4，0，4，…
- Cache采用直接映射组织

0 Miss

000	Mem(0)

4 Miss

000	Mem(0)
010	Mem(4)

0 Hit

000	Mem(0)
010	Mem(4)

4 Hit

000	Mem(0)
010	Mem(4)

- HR：0%！
  - 乒乓效应：由于映射冲突，导致cache块总是需要被替换

计算机组成与实现

组相联读写示例

- 假设cache只有4B，每块为1B
  - 各cache块初始均为空
- 假设cpu访存序列：0，4，0，4，…
- Cache采用2路组相联组织

0 Miss

000	Mem(0)

4 Miss

000	Mem(0)
010	Mem(4)

0 Hit

000	Mem(0)
010	Mem(4)

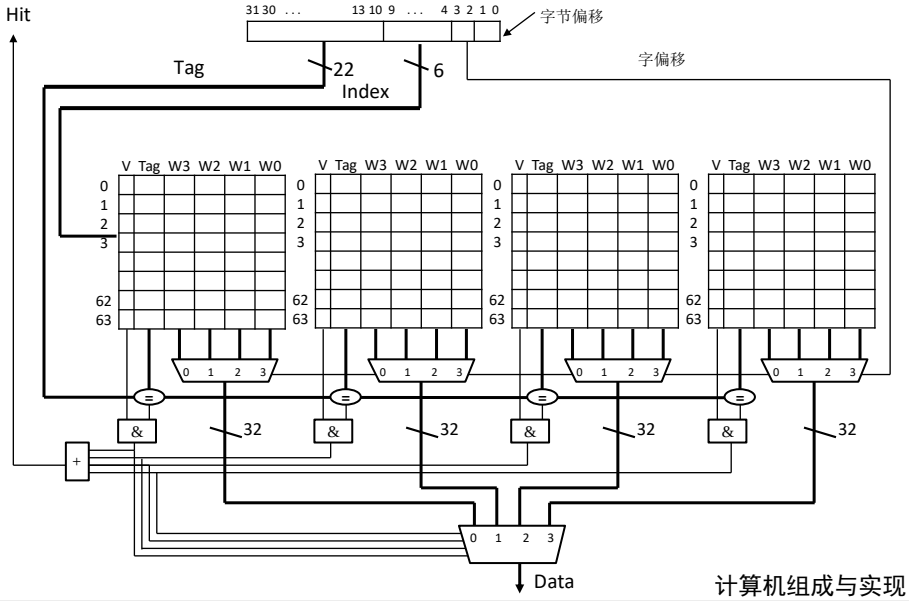
4 Hit

000	Mem(0)
010	Mem(4)

- HR：(n-2)/n。巨大的改善！
  - 由于一组内有多个cache块，提高了调度能力，于是降低了冲突

组相联cache内部结构

- 参数：32位主存地址；4路组相联，64组，块容量16B



组相联cache的实现成本

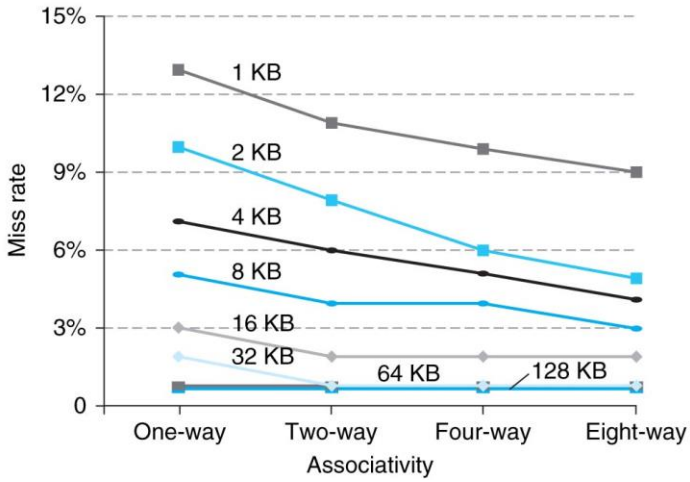
- N路组相联cache的主体是N个直接映射cache
  - 需要N个tag比较器
  - 需要MUX从N路数据中挑选正确的数据
- 当cache缺失时，存在一个问题：需要挑选组内哪个cache块？

Cache块替换策略

- 随机替换
  - 硬件随机的从组内挑选一个cache块
- 最近最少使用(Least Recently Used，LRU)
  - 硬件记录cpu的访存历史，然后替换最长时间未被访问的cache块
  - 对于2路组相联cache，每组设置一个1位的标记
- 由于LRU算法过于消耗资源，会导致性能下降。当4路以上时，通常会采用伪LRU算法

组相联cache的性能

- 性能最大改善发生在从直接映射改为2路组相联



计算机组成与实现

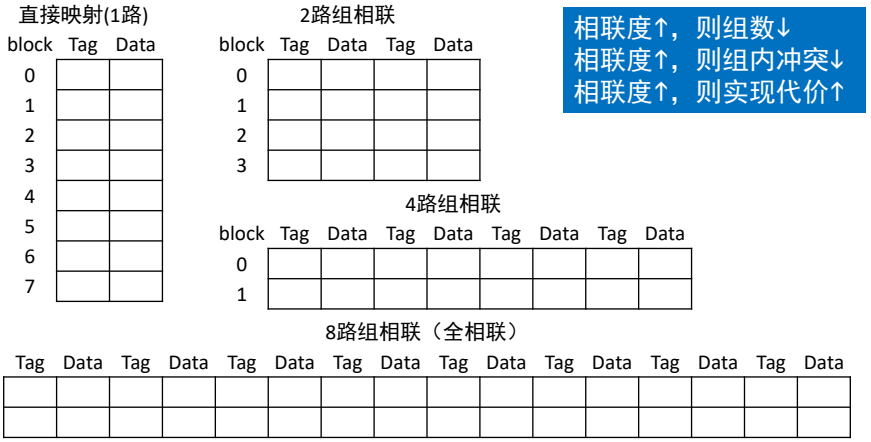
减少cache缺失

- 块替换策略对cache缺失率影响很大
- 直接映射：主存块只能映射到某个特定cache块
- 全相联：主存块可以映射到任意cache块
- N路组相联：将cache分成若干组，主存块可以映射到某个特定组内的任意cache块

计算机组成与实现

直接映射 vs 组相联 vs 全相联

- 直接映射与全相联是两种极端映射方案
- 组相联是两者的折中方案
- 示例：8个cache块的3种组织方案



相联度↑，则组数↓
相联度↑，则组内冲突↓
相联度↑，则实现代价↑

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

Cache设计考虑

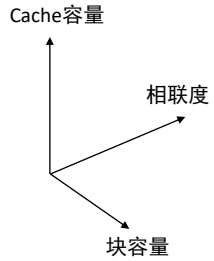
- L1$聚焦在缩短命中时间
  - 最小化HT来提升时钟频率
  - L1 MP要依靠L2$。所以，即使L1 MR高一些，也要追求快！
  - 通常容量都较小
- L2$、L3$聚焦在降低缺失率
  - 尽可能的避免访问主存（主存访问时间实在太慢！）
  - 通常容量较大，块尺寸也较大

如何改进cache性能？

- 减少cache的HT
  - 措施：更小的cache（查找/检查时间更短）
  - 示例：cache块只有1个字（不需要MUX！）
- 减少MR
  - 措施：更大的cache（capacity）
  - 措施：更大的块（有助于改善compulsory和空间局部性）
  - 措施：增加相联度（降低块内conflict）
- 减少MP
  - 措施：更小的块（减少数据copy时间）
  - 措施：使用多级cache
  - 措施：使用write buffer
    - 需要在cache读缺失时检查writebuffer

Cache设计空间

- Cache参数
  - Cache容量，块容量，相联度
- 策略
  - 写通vs.写回
  - 写分配vs.非写分配
  - 替换策略
- 优化方案的本质是折中
  - 取决于应用特性
    - 工作负载对于I$和D$的使用
  - 取决于技术/成本



简单往往就是最优！

提纲

- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

Multilevel Cache Practice (1/3)

- **Processor specs:**
  - CPI_{base} of 2
  - 100 cycle miss penalty to main memory
  - 25 cycle miss penalty to unified L2$
  - 36% of instructions are load/stores
  - 2% L1 I$ miss rate; 4% L1 D$ miss rate
  - 0.5% **global** U(nified)L2$ miss rate
- What is CPI_{stall} with and without the L2$?

Multilevel Cache Practice (2/3)

- **Notes:**
  - Both L1 I$ and L1 D$ send misses to L2$
  - What does the global L2$ MR mean?
    - MR to main memory
    - Since there are 2 L1$s, implies L2$ has 2 different local MRs depending on source of access
  - Use formula shown at bottom of slide 58

Multilevel Cache Practice (3/3)

- **Without L2$:**

Instr Fetch Load/Store

$$CPI_{stall} = 2 + 1\times0.02\times100 + 0.36\times0.04\times100$$

$$CPI_{stall} = 5.44$$

- **With L2$:**

$$CPI_{stall} = 2 + \boxed{1x.02\times25 \quad + .36\times.04\times25}\ \text{L1}$$

$$+ \boxed{1x.005\times100 + .36\times.005\times100}\ \text{L2}$$

$$= 3.54$$

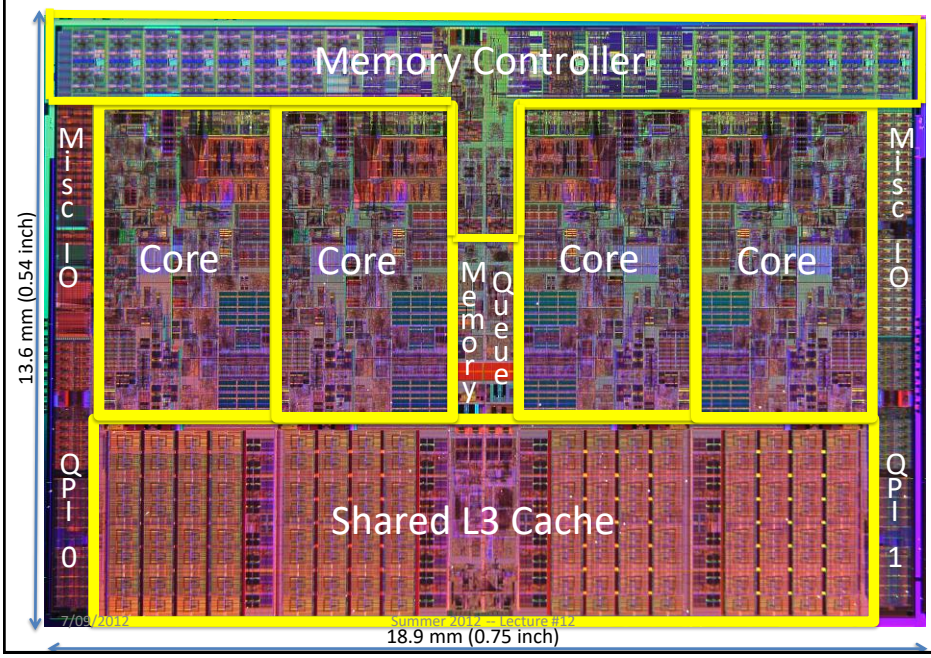
- CPI_{base} of 2
- 100 cycle miss penalty to main memory
- 25 cycle miss penalty to unified L2$
- 36% of instructions are load/stores
- 2% L1 I$ miss rate; 4% L1 D$ miss rate
- 0.5% **global** U(nified)L2$ miss rate

提纲

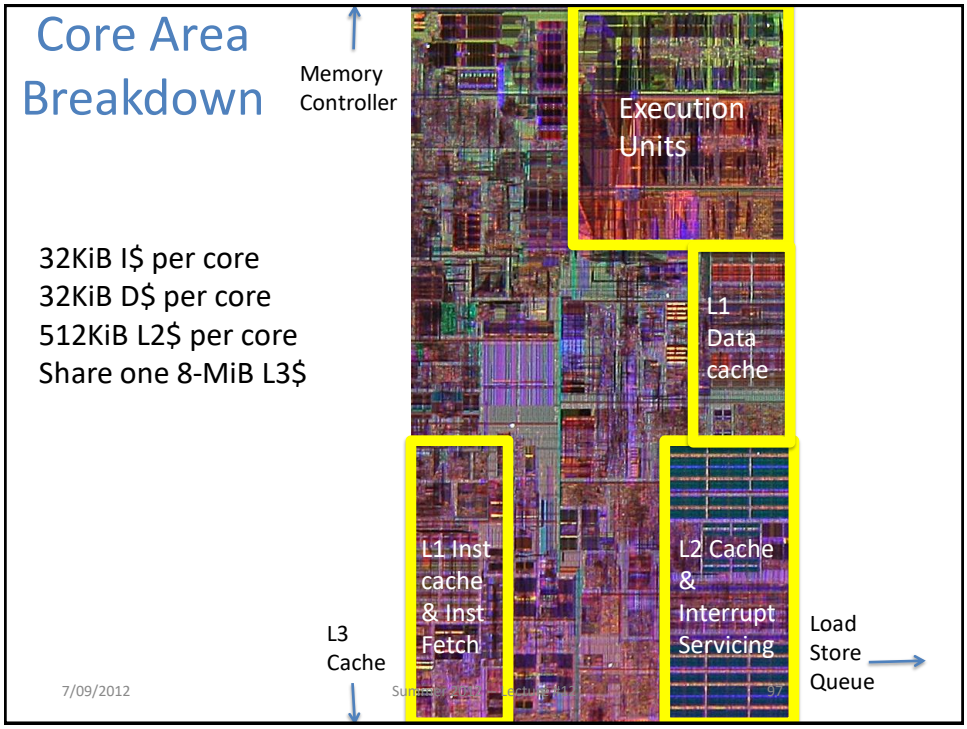
- 内容主要取材
  - CS61C的11讲和12讲
- 存储层次概述
- 直接映射Cache
- 直接映射Cache举例
- Cache读和写
- Cache性能
- 多级Cache
- 组相连Cache
- 改进Cache性能
- 多级Cache性能实战
- 当代Cache举例

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

Intel Nehalem Die Photo



Core Area Breakdown



32KiB I$ per core
32KiB D$ per core
512KiB L2$ per core
Share one 8-MiB L3$